Swift Format

Improving the Xcode Extension

CocoaHeads Montréal: May 16th 2019

Vincent Bernier

- Solutions Médias 360
- Company Goal: Modernizing the Car Dealer Industry
- Clients: Car Dealer Group & RDS
- Twitter: @2000Bernier

Searching For a Formatting Tool

Why were we caring for a Swift formatting tool?

- The team had a standard that was randomly enforced
- The desired standard was not possible with Xcode

What were we looking for?

- Tool that can apply the desired style

- Should be able to apply the formatting from Xcode (Xcode code extension)

What was out there

- Some had Xcode code extension
- All extension where bad or useless
- (I haven’t check recently so maybe there is some progress on that side)

SwiftFormat

Overview of 1 year ago

https://github.com/nicklockwood/SwiftFormat

Command Line Tool

- Format Swift Code
- Can include / exclude file and folders
- Can enable / disable rules
- Rules can be « tweak » by options
- Options can be infer from the content of file

Xcode Extension

- Format Swift Code
- ~~Can include / exclude file and folders~~
- ~~Can enable / disable rules~~
- ~~Rules can be « tweak » by options~~
- Options are inferred by the content of the file, with fallback on default when unable to figure it out

Xcode Extension: Goal

- Format Swift Code
- ~~Can include / exclude file and folders~~
- Can enable / disable rules
- Rules can be « tweak » by options
- ~~Instead of rules can infer code format base on the content of file~~
- ***Save settings to a file to share between team members***

Xcode Extension

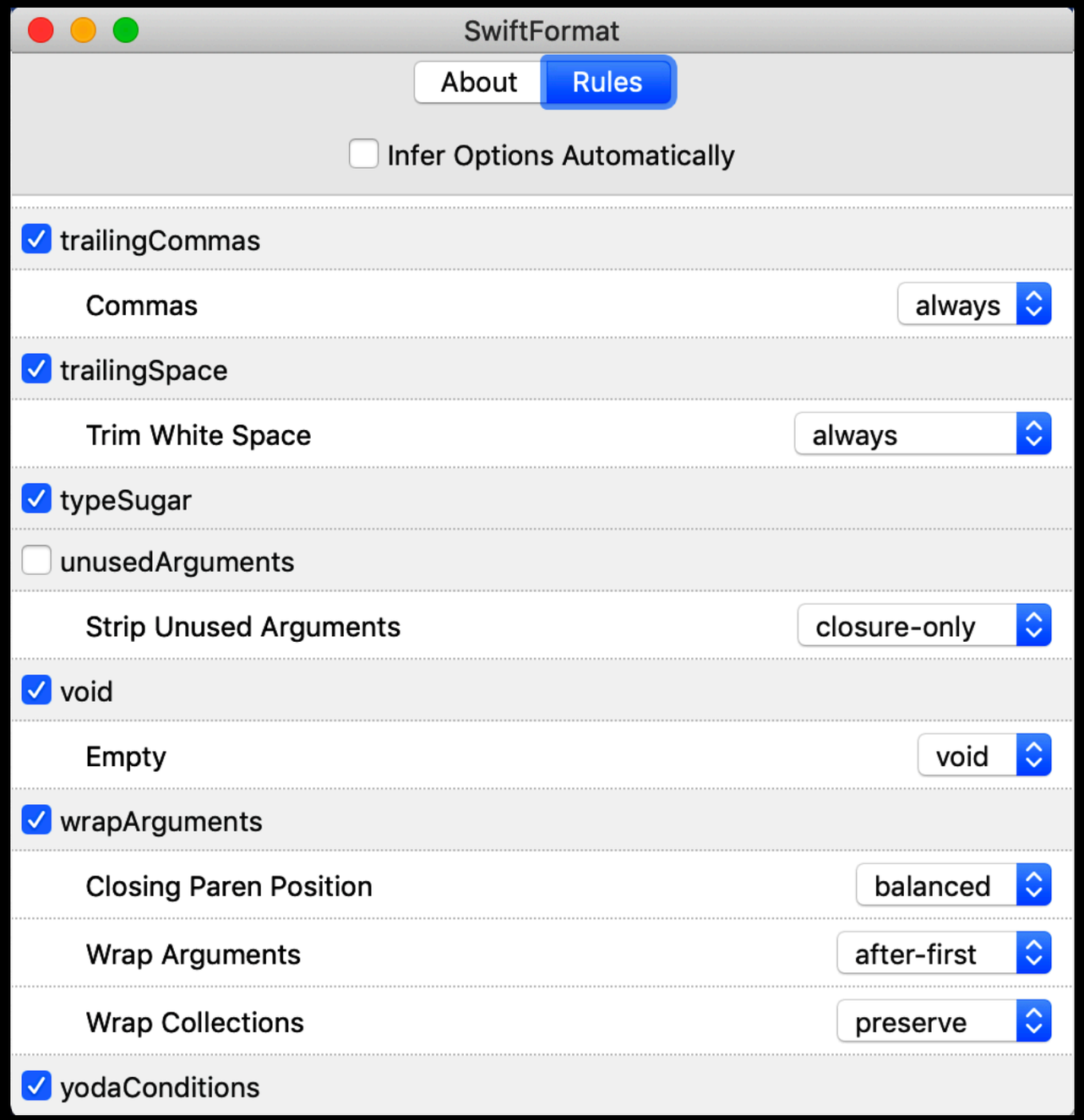
Code Extension Limitations

Limitation

- Install by a “companion” application via Preference System
- No access to file system (can’t read a config file)
- No configuration panel in Xcode (walk around…)

Goal

Have a way to configure



Iterate of Rules and Options

```
ruleStore
  .rules
  .sorted()
  .forEach { rule in
```

Options Agnostic UI

```
func tableView(_ tableView: NSTableView,
viewFor _: NSTableColumn?, row: Int) -> NSView? {

    let model = self.model(forRow: row)
    let id: NSUserInterfaceItemIdentifier
    switch model {
    case .binary:
        id = .binarySelectionTableCellView
    case .list:
        id = .listSelectionTableCellView
    case .freeText:
        id = .freeTextTableCellView
    }
    return tableView.makeView(withIdentifier: id,
                              owner: self)
}
```

Building It

Rules

- Rules had an array of names
- All was working with those names
- Rules can only be On / Off (binary)
- Just use the names

Options

- Options are not directly associated with rules in the code
- Options are property of a single object
- Options are parsed from the command line into the single object and back to command line in parsing code (huge monolith method)
- Houston, we have a problem…

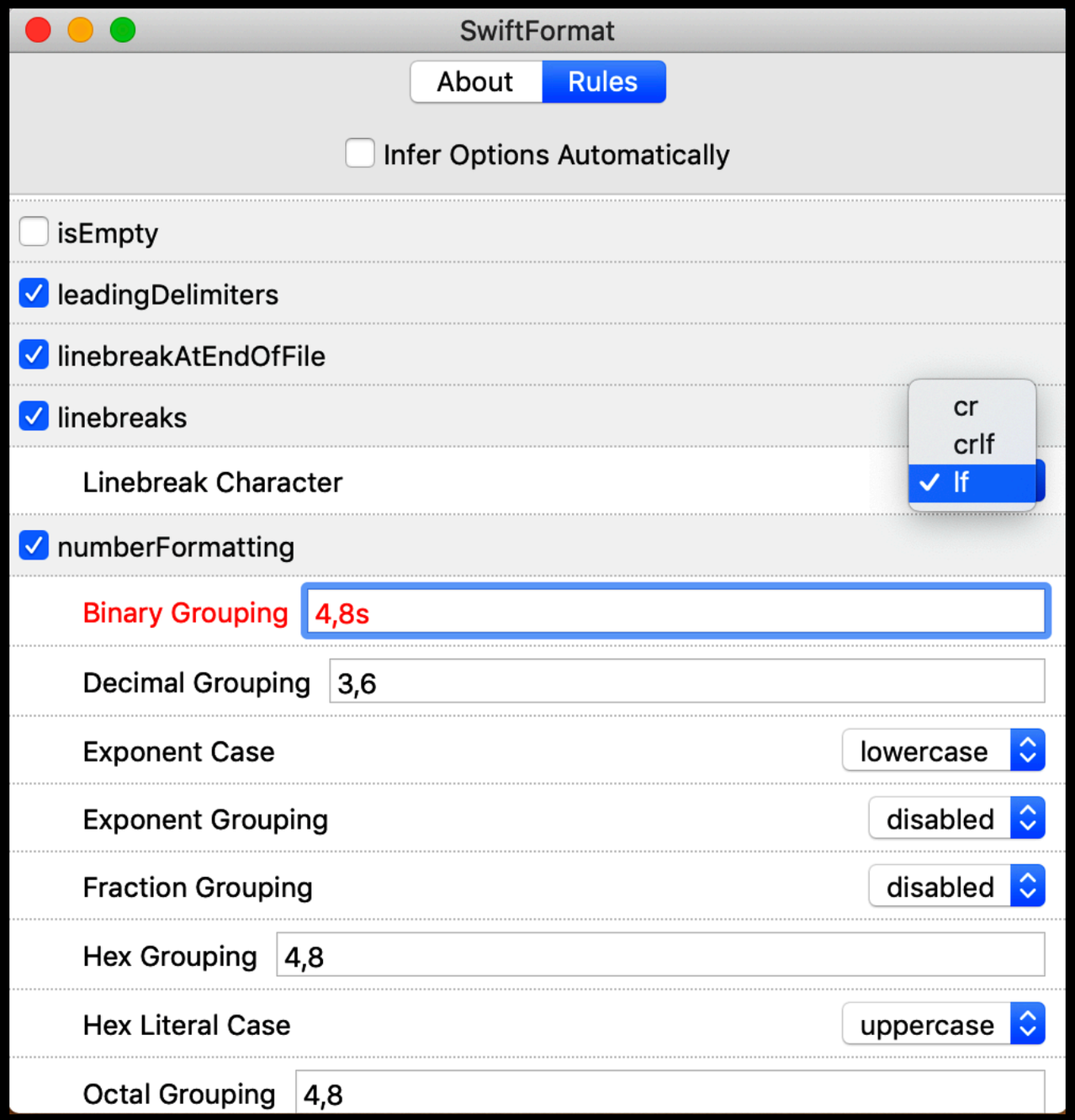
FormatOptions.Descriptor

```
extension FormatOptions {
struct Descriptor {

let id: String
let argumentName: String
let propertyName: String
let name: String
let type: ArgumentType
let defaultArgument: String

let toOptions:
  (String, inout FormatOptions) throws -> Void
let fromOptions: (FormatOptions) -> String
}}
```

Type of Options

```
enum ArgumentType: EnumAssociatable {
    case binary(true: [String], false: [String])
    case list([String])
    case freeText
           (validationStrategy: (String) -> Bool)
}
```

Argument Type



Argument Type

```
type: .freeText(validationStrategy: {
  Grouping(rawValue: $0) != nil
}),
```

```
type: .binary(true: ["uppercase", "upper"],
              false: ["lowercase", “lower”]),

type: .list(["unnamed-only",
             "closure-only",
             "always"]),
```

To Options

```
toOptions: { input, options in
  switch input.lowercased() {
  case "nextline", "next-line":
     options.elseOnNextLine = true
  case "sameline", "same-line":
     options.elseOnNextLine = false
  default:
     throw FormatError.options("")
  }
},
```

Options Processing

```
for opt in optionsToProcess {
  try processOption(opt.argumentName,
                    in: args,
                    from: &arguments,
                    to: &options,
                    handler: opt.toOptions)
}
```

From Options

```
fromOptions: { options in
 options.removeSelf ? "remove" : "insert"
}
```

```
fromOptions: { options in
  options.elseOnNextLine ?
                    "next-line" : "same-line"
}

fromOptions: { options in
  options.stripUnusedArguments.rawValue
}
```

There is Unit Test in the Project

A lot of duplication

- There was only 3 type of Options
- Different option type have a lot of property in commons
- Same type have similar transformation

Simple Validation

```
validateSut(sut,
            id: "void-representation",
            name: "empty",
            argumentName: “empty",
            propertyName: “useVoid")

validateSutThrowFormatErrorOptions(sut)

validateArgumentsBinaryType(
          sut: sut,
          controlTrue: ["void"],
          controlFalse: ["tuple", “tuples"],
          default: true)
```

What About Transformation

- Each test need to target a different property

KeyPath & Generic

```
func validateFromArguments<T: Equatable>(
    sut: FormatOptions.Descriptor,
    keyPath: WritableKeyPath<FormatOptions, T>,
    expectations: [OptionArgumentMapping<T>],
    testName: String = #function)

//=========
let expectationsExample:
      [OptionArgumentMapping<Bool>] = [
          (optionValue: false,
                argumentValue: "tuple"),
          (optionValue: true,
                argumentValue: "void"),
      ]
```

KeyPath & Generic

```
validateFromOptions(
                sut: sut,
                keyPath: \FormatOptions.useVoid,
                expectations: fromOptionsExpectation)
```

```
validateFromArgumentsBinaryType(
                    sut: sut,
                    keyPath: \FormatOptions.useVoid)

validateFromArguments(
                sut: sut,
                keyPath: \FormatOptions.wrapArguments,
                expectations: expectedMapping)
```

Closer Look

at KeyPath

5 Classes

- AnyKeyPath
- PartialKeyPath<Root>
- KeyPath<Root, Value>
- WritableKeyPath<Root, Value>
- ReferenceWritableKeyPath<Root, Value>

KeyPath<Root, Value>

```
// KeyPath<URL, String>
let readonlyOnStruct = \URL.absoluteString

// KeyPath<MyClass, String>
let constantOnReferenceType =
                          \MyClass.letConstant

// KeyPath<MyClass, NSViewController>
let constantRefOnReferenceType =
                          \MyClass.viewController
```

WritableKeyPath

```
// WritableKeyPath<URLRequest, String?>
let readWriteOnStruct = \URLRequest.httpMethod

// ReferenceWritableKeyPath
        <NSViewController, [NSViewController]>
let readWriteOnReferenceType =
                    \NSViewController.children

// ReferenceWritableKeyPath
          <MyClass, [NSViewController]>
let chainingPath =
                \MyClass.viewController.children
```

Usage

With a named subscript[keyPath: …]

object[keyPath: aKeyPath] = newValue

Explaining with Code

```
struct AsProperty {
  var name: String?
}
struct MyStruct {
  var asProperty = AsProperty()
  var title: String?
}

func setString<T>(
          _ str: String,
          on object: inout T,
          at keyPath: WritableKeyPath<T, String?>)
{
  object[keyPath: keyPath] = str
}
```

Explaining with Code

```
var myStruct = MyStruct()

// <MyStruct, String?>
setString("out",
          on: &myStruct,
          at: \MyStruct.title)

// Same root and same value <MyStruct, String?>
setString("in",
          on: &myStruct,
          at: \MyStruct.asProperty.name)

// different root <AsProperty, String?>
setString("name",
          on: &myStruct.asProperty,
          at: \AsProperty.name)
```

Wrap Up

Conclusion

- Descriptor allowed to build the table representation with a for loop
- Then build the UI with a simple switch statement
- KeyPath allowed to reduce code duplication in Test
- And afterward to reduce code duplication in OptionDescriptor transformation code

Questions

The End